CSE 505 Assignment 4 Solution and Grading Guide

Andy Collins
acollins@cs.washington.edu

December 10, 2003

Point distribution

Yet again, this homework is graded out of 30 points, to make it equal in value to the others, even though it is shorter. The points were divided 8, 8, and 14 to questions 1, 2, and 3, respectively. (1a) and (1b) were worth three points each, and all other parts two points, except (3e), which was worth six points.

Overall, scores were quite good, and it looked like most everybody understood what was going on. By and large this was the main grading criterion, especially for problem 2, where we hadn't necessarily fully developed all of the language to talk about these issues.

1 Typed currying

Nearly everybody did well here. The O'Caml implementations were universally correct, and the errors in the System F expressions all looked minor and typo-ish. I saw lots of ways of implementing `not-curry` and `not-uncurry`, and I suspect some people were just trying to keep my life interesting with their creativity here.

1.1 in System F

$$e_1 = \Lambda\alpha_1.\ \Lambda\alpha_2.\ \Lambda\alpha_3.\ \lambda x_1{:}(\alpha_1 * \alpha_2) \to \alpha_3.\ \lambda x_2{:}\alpha_1.\ \lambda x_3{:}\alpha_2.\ x_1\ (x_2, x_3)$$

$$e_2 = \Lambda\alpha_1.\ \Lambda\alpha_2.\ \Lambda\alpha_3.\ \lambda x_1{:}\alpha_1 \to \alpha_2 \to \alpha_3.\ \lambda x_2{:}\alpha_1 * \alpha_2.\ x_1\ x_2.1\ x_2.2$$

1.2 in O'Caml

The basic solution to this in O'Caml is something like:

```
let curry   f = fun x -> fun y -> f (x,y)
let uncurry f = fun (x,y) -> f x y
```

using explicit `fun` constructs to build a curried function. Of course you can also use O'Caml syntactic sugar; the shortest solution I saw was:

```
let curry f x y = f (x,y)
let uncurry f (x,y) = f x y
```

1.3 not in O'Caml

The point here was that O'Caml can use mutation, exceptions, and/or infinite loops to create something that looks like a curry or uncurry function, but is not. This means that our "free theorems" do not translate to the O'Caml world, because they explicitly assume no side effects. Infinite loops are an interesting and more arguable case. I let them go here, although one could make the argument that a function that never

terminates *is* a curry or uncurry (if the type is right), because it never returns anything that isn't an appropriately curried or uncurried output. This goes back to our discussion of what means "equivalent" in lecture.

The following is an implementation with exceptions (which was the most common solution, and probably the most simple and brief):

```
exception E

let not_curry   f = (raise E; curry f)
let not_uncurry f = (raise E; uncurry f)
```

2 Picking on Java

Dan's concise answers to the four parts are:

(a) In main, the second argument (`a`) in the call to `f` subsumes `C[]` to `Object[]`. Java's rule for subtyping arrays is `C[]<D[]` if `C<D`.

(b) The program does *not* execute any downcasts, implicit or explict. Running it causes an uncaught ArrayStoreException to be thrown.

(c) Array update takes an array-object a, an index i, and on object o. The array-object a has a length and an element-type, both chosen when the object is constructed. For example, the expression `new C[10]` constructs an array with length 10 and element-type `C`. If i is greater than the length, an ArrayBoundsException is thrown. If o has a different run-time type than a's element type, an ArrayStoreException is thrown. Else, the i^{th} element of the array is mutated to refer to o.

(d) No. Implementing array-update requires, at run-time, the type of the array's elements and the type of the object being assigned to an array element. (This would not be necessary if array subtyping was invariant!)

To which I might add some commentary:

For part (a), there are two things I wanted to see: something to tell me that the subsumption happens at the point where the function `f` is called, and a description of the generic subtyping rule. We don't care about the overall typing rule for arrays (which basically says that something has type `C[]` if its elements have type `C`, only the subtyping rule to relate two arrays.

For part (b), I was looking for recognition that the statement `arr[0] = x;` cannot be executed and throws an exception, and some explanation of why it cannot be executed. I didn't need to see, although you should definitely understand, why we got into this situation where the type system was insufficient to recognize that we were going to get into trouble. We can argue as to whether the operation of checking to see whether the runtime type of the object to be put in the array is a subtype of the contents of the runtime type of the array constitutes a "implicit downcast," but ultimately Dan thinks it does, and he has the biggest vote. My concern was that you knew what was happening, rather than worrying about what to call it.

Everybody rightly focused on the error conditions for part (c). I didn't worry too much about the formalism for describing what an array is and what it means to update it, because we all know what it means to update the i'th element of an array, and we haven't carefully delineated a formal model in class (although we certainly could—we do have the mathematical machinery to do so if we wanted). I also focused on the description of the `ArrayStoreException` rule, because that is the main point of the question.

The critical thing to recognize for part (c) is that the example program itself is a perfect example of why we need runtime type information. Without it we could not implement the proper Java error behavior, so it isn't even necessary to exhibit a non-error piece of code to illustrate the requirement. It's just as important when implementing a language to adhere to the rules for error conditions as for correct programs.

3 Strong evaluator interfaces

(a)
```
let interpret1 e =
  let _ = typecheck mt_ctxt e in
  interpret mt_env e
```

(b)
```
let state = ref []
let was_checked e =
  let rec f l =
    match l with
      [] -> false
    | hd::tl -> e==hd || f tl in
  f (!state)

let typecheck2 e =
  let ans = typecheck mt_ctxt e in
  state := e::(!state);
  ans

let interpret2 e =
  if was_checked e
  then interpret mt_env e
  else raise TypeError
```

(c)
```
let typecheck3 e =
  let _ = typecheck mt_ctxt e in
  fun () -> interpret mt_env e
```

(d)
```
let typecheck4 e =
  let _ = typecheck mt_ctxt e in
  e

let interpret4 = interpret mt_env
```

(e) Only `interpret1` is still safe. It is impossible for a client to mutate the expression in-between typechecking and evaluation because control remains in the implementation. (By the way, it would likely be unsafe in a shared-data preemptive multithreading situation.)

The others are all unsafe for the same basic reason, and the exploits are all essentially the same. Typecheck one piece of code, mutate it somewhere inside the AST (which doesn't change pointer-equality for the root of the AST), and then execute the code as required by the interface. Thus the list of checked programs in (b) is fooled because it uses pointer equality; the thunk in (c) doesn't help because the type-checked program bound up inside the thunk can still be mutated from outside through the pointer; and the typesystem trick in (d) doesn't help because the mutation is happening at the level below the typesystem, where the object of type `tcecp` really is the same AST that can be mutated.

Note that we could build an interface that would work, but we would need to copy the AST into private memory (like an OS kernel might do), and give the client only a handle that is not a true pointer.

Dan's `adversary.ml` looks like:

```
open Ast2

let make_app () = Apply(ref (Fun(``x'',Bool,Var ``x'')), True)
let change_app e =
  match e with
```

```
    Apply(x,_) -> x := True
  | _ -> ()

let break_b () =
  let e = make_app () in
  ignore(Impl2.typecheck2 e);
  change_app e;
  Impl2.interpret2 e

let break_c () =
  let e = make_app () in
  let f = Impl2.typecheck3 e in
  change_app e;
  f ()

let break_d () =
  let e  = make_app () in
  let e2 = Impl2.typecheck4 e in
  change_app e;
  Impl2.interpret4 e2

let _ = break_b()
```

This particular version does not try to catch the exceptions and turn them into printed messages, so it can only test one exploit at a time. You select which one by choosing one of the `break_[b-d]` functions in the last line.

Note that I didn't actually compile or run any of this code, which means that I didn't worry about whether your `adversary.ml` required any commenting/uncommenting to avoid name collisions when testing the different parts. I basically just looked to see the form of the exploits.